Making Higher-Order Superposition Work

Petar Vukmirović[1], Alexander Bentkamp[1], Jasmin Blanchette[1,2,3], Simon Cruanes[4], Visa Nummelin[1], and Sophie Tourret[2,3]

[1] Vrije Universiteit Amsterdam, Amsterdam, the Netherlands
{p.vukmirovic,a.bentkamp,j.c.blanchette,visa.nummelin}@vu.nl
[2] Université de Lorraine, CNRS, Inria, LORIA, Nancy, France
sophie.tourret@inria.fr
[3] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[4] Aesthetic Integration, Austin, Texas, USA
simon@imandra.ai

Abstract. Superposition is among the most successful calculi for first-order logic. Its extension to higher-order logic introduces new challenges such as infinitely branching inference rules, new possibilities such as native reasoning with formulas, and the need to curb the explosion of specific higher-order rules. We describe techniques that address these issues and extensively evaluate their implementation in the Zipperposition theorem prover. Largely thanks to them, Zipperposition won the higher-order division of the CASC-J10 competition.

1 Introduction

In the last decades, superposition-based first-order automatic theorem provers have emerged as useful reasoning tools. They dominate at the annual CASC [40] theorem prover competitions, winning the first-order theorem division since the competition's inception. They are also used as backends to proof assistants [12, 22, 31], automatic higher-order theorem provers [37], and software verifiers [14]. The superposition calculus has only recently been extended to higher-order logic, resulting in λ-superposition [3], which we developed together with Waldmann, as well as combinatory superposition [8] by Bhayat and Reger.

Both of these higher-order superposition calculi were designed to gracefully extend first-order reasoning. As it is expected that most steps in higher-order proofs are essentially first-order, extending the most successful first-order calculus to higher-order logic seemed like a good strategy to improve the state of the art. Our first attempt at corroborating this conjecture was in 2019: Zipperposition 1.5, which implemented λ-superposition, finished third in the higher-order theorem division of CASC-27 [42], 12 percentage points behind the winner, Satallax 3.4, based on tableaux and SAT solving.

Investigating these results, we discovered that tableaux have some advantages over higher-order superposition. To mitigate this, we developed techniques and heuristics to simulate the behavior of a tableau prover in a saturating prover. We implemented them in Zipperposition 2, which took part in CASC-J10 [43] one year later. This time, Zipperposition won the division, solving 84% of problems, a whole 20 percentage points ahead of the next best prover, Satallax 3.4. In this paper, we describe the main techniques that explain this reversal of fortunes. They range from preprocessing to backend integration.

Interesting patterns can be observed in various higher-order encodings of problems. We show how we can exploit them to simplify problems (Sect. 3). By working on formulas rather than clauses, tableau techniques take a more holistic view at a higher-order problem. Delaying the clausification through the use of calculus rules that act on formulas achieves the same effect in superposition. We further explore the benefits of this approach (Sect. 4).

The main drawback of λ-superposition compared with combinatory superposition is that it relies on rules that enumerate possibly infinite sets of unifiers. We describe a mechanism that interleaves performing infinitely branching inferences with the standard saturation process (Sect. 5). It allows the prover to retain the same behavior as before on first-order problems, smoothly scaling with increased number of higher-order clauses. We also propose some heuristics to curb the explosion induced by highly prolific λ-superposition rules (Sect. 6).

Using first-order backends to finish the proof attempt is common practice in higher-order reasoning. Since λ-superposition behaves like regular superposition on the first-order part of the problem, invoking backends may seem redundant and pointless; yet Zipperposition is nowhere as efficient as E [33] or Vampire [25], so invoking a more efficient backend does make sense. We describe how to achieve a balance between allowing the higher-order calculus to do native reasoning and delegating reasoning to a backend (Sect. 7).

We also compare Zipperposition 2 with other provers on all monomorphic higher-order TPTP benchmarks [41] to perform a more extensive evaluation than at CASC (Sect. 8). Our evaluation corroborates the competition results.

2 Background and Setting

Our setting is monomorphic higher-order logic, but the techniques can be extended with polymorphism. Indeed, our implementation is polymorphic.

Higher-Order Logic. We define terms inductively as free variables F, G, X, Y, bound variables x, y, z, constants $\mathsf{f}, \mathsf{g}, \mathsf{a}, \mathsf{b}, \ldots$, applications $s\,t$, or λ-abstractions $\lambda x.\, s$, where s and t are terms. The syntactic distinction between free and bound variables gives rise to *loose bound variables* (e.g., y in $\lambda x.\, y\,\mathsf{a}$) [27]. We write $s\,\bar{t}_n$ to denote $s\,t_1 \cdots t_n$ and $\lambda \overline{x}_n.\, s$ for $\lambda x_1. \cdots \lambda x_n.\, s$. Every β-normal term can be written as $\lambda \overline{x}_m.\, s\,\bar{t}_n$, where s is not an application; we call s the *head* of the term. If s is a variable of a type of the form $\tau_1 \to \cdots \to \tau_n \to o$, where o is the distinguished Boolean type and $n \geq 0$, we call s a *predicate variable*. A literal l is an equation $s \approx t$ or a disequation $s \not\approx t$. A clause is a finite multiset of literals, interpreted and written disjunctively $l_1 \vee \cdots \vee l_n$. Term-building logical symbols are written in boldface: $\boldsymbol{\neg}, \boldsymbol{\wedge}, \boldsymbol{\vee}, \boldsymbol{\rightarrow}, \boldsymbol{\leftrightarrow}, \ldots$ Nonequational (predicate) literals are encoded as (dis)equations with $\mathbf{\top}$ based on their sign: For example, $\mathsf{even}(x)$ becomes $\mathsf{even}(x) \approx \mathbf{\top}$, and $\neg\,\mathsf{even}(x)$ becomes $\mathsf{even}(x) \not\approx \mathbf{\top}$.

Higher-Order Calculi. The λ-superposition calculus is a refutationally complete inference system and redundancy criterion for Boolean-free extensional polymorphic clausal higher-order logic. A pragmatic, possibly incomplete extension of this calculus to higher-order logic with Booleans is described by Vukmirović and Nummelin [47]. The λ-superposition calculus relies on *complete sets of unifiers* (*CSUs*). The CSU for s and t, with respect to a set of variables V, denoted by $\mathrm{CSU}_V(s,t)$, is a set of unifiers such that for any unifier ϱ of s and t, there exists a $\sigma \in \mathrm{CSU}_V(s,t)$ and θ such that $\varrho(X) = (\sigma \circ \theta)(X)$ for all $X \in V$. The set X is used to distinguish between important and auxiliary variables. We can normally leave it implicit.

By contrast, the combinatory superposition calculus avoids CSUs by using a form of first-order unification, but it enumerates higher-order terms using rules that instantiate applied variables with partially applied combinators from the complete combinator set $\{\mathsf{S}, \mathsf{K}, \mathsf{B}, \mathsf{C}, \mathsf{I}\}$. This calculus is the basis of Vampire 4.5 [8], which finished closely behind Satallax 3.4 at CASC-J10.

A different, very successful calculus is Satallax's SAT-guided tableaux [1]. Satallax was the leading higher-order prover of the 2010s. Its simple and elegant tableaux avoid deep superposition-style rewriting inferences. Nevertheless, our working hypothesis for the past six years has been that superposition, having proved its value for first-order logic with equality already in the 1990s, would likely provide a stronger basis for higher-order reasoning. Other competing higher-order calculi include SMT (implemented in CVC4 [2]) and extensional paramodulation (implemented in Leo-III [37]).

Zipperposition. Zipperposition [3, 11] is a higher-order theorem prover based on a pragmatic extension of λ-superposition. It was conceived as a testbed for rapid experimenting with extensions of first-order superposition, but over time, it has assimilated most of the techniques and heuristics of the E prover [33]. Zipperposition 2 also implements combinatory superposition.

Several of our techniques extend the *given clause procedure* [26, Section 2.3], the standard saturation procedure. It partitions the proof state into a set P of *passive* clauses and a set A of *active* clauses. Initially, P contains all input clauses, and A is empty. At each iteration, a *given* clause C from P is moved to A (i.e., it is *activated*), all inferences between C and clauses in A are performed, and the conclusions are added to P.

Experimental Setup. To assess the practical value of our proposed techniques, we carried out experiments with Zipperposition 2. As benchmarks, we use all 2606 monomorphic higher-order problems from the TPTP library [41] version 7.2.0. We fixed a *base* configuration of Zipperposition parameters as a baseline for all comparisons. Then, in each experiment, we vary the parameters associated with a specific technique to evaluate it. The experiments were performed on StarExec [38] servers, which are equipped with Intel Xeon E5-2609 CPUs clocked at 2.40 GHz. Unless otherwise stated, we used a CPU time limit of 20 s, which is roughly the time each configuration is given in the portfolio mode used for CASC.

3 Preprocessing Higher-Order Problems

The TPTP library contains thousands of higher-order problems. Despite their diversity, they have a markedly different flavor from the TPTP first-order problems. Notably, they extensively use the `definition` role to identify universally quantified equations (or equivalences) that define symbols. For some problems, eagerly unfolding all the definitions and β-reducing the results eliminates all higher-order features, making the problem amenable to first-order methods. But often, unfolding the definitions inflates the problem beyond recognition. Therefore, higher-order provers need to be flexible.

An effective way to deal with definitions is to interpret them as rewrite rules, using the orientation given in the input problem. If there are multiple definitions for the same symbol, only the first one is used as a rewrite rule. Then, whenever a clause is picked in the given clause procedure, it will be rewritten using the collected rules. Since the TPTP format enforces no constraints on definitions, this rewriting might not terminate. To ensure termination, we limit the number of applied rewrite steps. In practice, most TPTP problems are well behaved: Only one definition per symbol is given, and the definitions are acyclic. Instead of rewriting a clause when it is activated, we can rewrite the input formulas as a preprocessing step. This has the advantage that input clauses will be fully simplified when the proving process starts, and no defined symbols will occur in clauses, which usually helps the heuristics.

Rewriting definitions in this way can compromise the refutational completeness of the superposition calculi. The calculus allows simplifications that lead to smaller clauses according to a fixed term order lifted to clauses, but often unfolding a definition will lead to a larger or incomparable clause. Typically, the Knuth–Bendix order (KBO) [23] is used. It compares terms by first comparing their weights, which is the sum of all the weights assigned to the symbols it contains. To recover completeness, we do the following. Given a symbol weight assignment $\mathcal{W}$, we can update it so that it orients acyclic definitions from left to right assuming that they are of the form $\mathsf{f}\,\overline{X}_m \approx \lambda\overline{y}_n.\, t$, where the only free variables in t are $\overline{X}_m$, no free variable repeats or appears applied in t, and f does not occur in t. Then we traverse the symbols f that are defined by such equations following the dependency relation, starting with a symbol f that does not depend on any other defined symbol. For each f, we set $\mathcal{W}(\mathsf{f})$ to $w+1$, where w is the maximum weight of the right-hand sides of f's definitions, computed using $\mathcal{W}$. By construction, each left-hand side is heavier than the corresponding right-hand side, thereby justifying a left-to-right orientation.

Evaluation and Discussion. Axioms annotated with `definition` are treated as rewrite rules in *base*, and it preprocesses the formulas using the rewrite rules. We also tested the effects of disabling this preprocessing ($-$preprocess), disabling the special treatment of `definition` axioms ($-$RW), and disabling the special treatment of `definition` while using adjusted KBO weight as descibed above ($-$RW+KBO). The results are given in Figure 1. In all of the figures in this paper, each cell gives the number of proved problems; the highest number is typeset

base	−preprocess	−RW	−RW+KBO
1638	1627	1303	1324

Fig. 1: Effect of the rewrite methods

	+A	−A
MC	1624	1638
DCI	1496	1531
DCS	1659	**1710**

Fig. 2: Effect of clausification and AVATAR

in bold. Clearly, treating `definition` axioms as rewrite rules greatly improves performance. Using adjusted KBO weight is not as strong, but raw evaluation data reveals that it yields 15 solutions not found using other configurations.

4 Reasoning with Formulas

Higher-order logic identifies terms and formulas. In many cases, we can easily solve a problem by instantiating a predicate variable with the right formula. Finding this formula is usually easier if the problem is not clausified. Consider the conjecture $\exists f.\, f\,\mathsf{p}\,\mathsf{q} \leftrightarrow \mathsf{p} \wedge \mathsf{q}$. Expressed in this form, the formula is easily to prove by taking $f := \lambda x\, y.\, x \wedge y$. By contrast, guessing the right instantiation for the negated, clausified form $F\,\mathsf{p}\,\mathsf{q} \not\approx \top \vee \mathsf{p} \not\approx \top \vee \mathsf{q} \not\approx \top, F\,\mathsf{p}\,\mathsf{q} \approx \top \vee \mathsf{p} \approx \top$, $F\,\mathsf{p}\,\mathsf{q} \approx \top \vee \mathsf{q} \approx \top$ is more challenging. One of the strengths of tableau provers is that they do not clausify the input problem. This might explain Satallax's dominance in the THF division of CASC competitions until CASC-J10.

This example shows that new formulas can appear during proof search. Thus, clausal higher-order calculi typically need some form of calculus-level clausification. We previously studied such clausification techniques in both ad hoc [47] and complete [4] extensions of λ-superposition. Both approaches feature *dynamic clausification rules* that incrementally clausify top-level logical symbols; for example, a clause $C' \vee (\mathsf{p} \wedge \mathsf{q}) \not\approx \top$ gives rise to new clause $C' \vee \mathsf{p} \not\approx \top \vee \mathsf{q} \not\approx \top$.

Dynamic clausification rules can be used as inference rules (which add conclusions to the passive set) or as simplification rules (which delete premises and add conclusions to the passive set). Inferences are more flexible as they produce all intermediate clausification states, whereas simplifications produce fewer clauses. Since clausifying equivalences can destroy a lot of syntactic structure [15], we clausify them using inferences.

We discuss two tableaux-inspired approaches to reasoning with formulas. First, we study the interference of clause-splitting techniques with dynamic clausification. Second, we show how to perform useful quantifier instantiations in a saturating prover.

Some superposition provers, including Zipperposition, support variants of AVATAR [45], an architecture that makes it possible to partition the search space by splitting clauses into variable-disjoint subclauses. Combining AVATAR and dynamic clausification makes it possible to split a higher-order clause $(\varphi_1 \vee$

$\cdots \vee \varphi_n) \approx \top$, where φ_i are arbitrarily complex formulas that mutually share no free variables, into clauses $\varphi_i \approx \top$.

To finish the proof, it suffices to derive $\bot$ under each assumption $\varphi_i \approx \top$. As the split is performed on the formula level, this technique resembles tableaux, but it exploits the strengths of superposition, such as its powerful redundancy criterion and simplification machinery, to close the branches.

When dynamic clausification is used, the prover can make heuristic choices based on information obtained from the formulas on which clausification rules are applied. In particular, we look for λ-abstractions whose bodies are formulas in each active clause and store them in a set *Inst*. Optionally, *Inst* can contain *primitive instantiations* [47]—that is, imitations (in the sense of higher-order unification) of logical symbols that approximate the shape of a formulas that can instantiate a predicate variable. These instantiations resemble the ones tableau provers perform during the proof search.

Whenever we use dynamic clausification to replace the predicate variable x in a clause of the form $(\forall x.\, \varphi) \approx \top \vee C$ with a fresh variable X, resulting in $\varphi\{x \mapsto X\} \approx \top \vee C$, we create additional clauses in which x is replaced with each term $t \in \mathit{Inst}$ of the same type as x. However, as a new term t can be added to *Inst* after a clause with a quantified variable of the same type as t has been activated, we must also keep track of the clauses $\varphi\{x \mapsto X\} \approx \top \vee C$, so that when *Inst* is extended, we instantiate the saved clauses. Conveniently, instantiated clauses are not recognized as redundant with respect to the original clause since Zipperposition uses an optimized but incomplete subsumption algorithm.

Given a disequation $\mathsf{f}\,\bar{s}_n \not\approx \mathsf{f}\,\bar{t}_n$, the *abstraction* of s_i is $\lambda x.\, u \approx v$, where u is the result of replacing s_i by x in $\mathsf{f}\,\bar{s}_n$ and v is the result of replacing s_i by x in $\mathsf{f}\,\bar{t}_n$ (if it occurs in $\mathsf{f}\,\bar{t}_n$). For an equation $\mathsf{f}\,\bar{s}_n \approx \mathsf{f}\,\bar{t}_n$, the analogous abstraction is $\lambda x.\, \neg\,(u \approx v)$.

We have observed that adding abstractions of the literals in the conjecture to *Inst* provides useful instantiations for formulas such as induction principles for datatypes. Consider the TPTP problem `DAT056^2` [39], whose clausified conjecture is $\mathsf{ap}\,\mathsf{xs}\,(\mathsf{ap}\,\mathsf{ys}\,\mathsf{zs}) \not\approx \mathsf{ap}\,(\mathsf{ap}\,\mathsf{xs}\,\mathsf{ys})\,\mathsf{zs}$, where ap is the list append operator defined recursively on its first argument and xs, ys, and zs are of list type. Abstraction of xs from the disequation yields $t = \lambda x.\, \mathsf{ap}\,x\,(\mathsf{ap}\,\mathsf{ys}\,\mathsf{zs}) \approx \mathsf{ap}\,(\mathsf{ap}\,x\,\mathsf{ys})\,\mathsf{zs}$, which is inserted into *Inst*. One of the axioms in the problem is the induction axiom for the list datatype: $\forall p.\, (p\,\mathsf{nil} \wedge (\forall x\, xs.\, p\, xs \rightarrow p\,(\mathsf{cons}\,x\,xs))) \rightarrow \forall xs.\, p\,xs$, where nil and cons have the usual meanings. Instantiating p in this axiom with the abstraction t, and the equations defining ap allow us to prove $\forall x.\, \mathsf{ap}\,x\,(\mathsf{ap}\,\mathsf{ys}\,\mathsf{zs}) \approx \mathsf{ap}\,(\mathsf{ap}\,x\,\mathsf{ys})\,\mathsf{zs}$, from which we can easily derive a contradiction.

Evaluation and Discussion. The base configuration uses *monolithic clausification* (MC), an approach that applies a standard clausification algorithm [29] both as a preprocessing step and after instantiating predicate variables. Zipperposition supports only a limited version of AVATAR [45], which is disabled in the base configuration. To test the merits of dynamic clausification, we change *base* along two axes: We choose monolithic clausification (MC), dynamic clausification as inference (DCI), or dynamic clausification as simplification (DCS), and we either enable (+A) or disable (−A) the lightweight AVATAR. The base configuration does not use Boolean instantiation.

Figure 2 shows that using simplifying dynamic clausification greatly increases the success rate, while using dynamic clausification as an inference has the opposite effect. Manually inspecting the proofs found by DCS configuration, we found out that not clausifying equivalences was one of the main reasons behind strong performance of this configuration.

Overall, the lightweight AVATAR harms performance, but the sets of problems solved with and without AVATAR are vastly different. For example, the MC+A configuration solves 60 problems not solved by MC−A.

The Boolean instantiation technique described above is applicable only when dynamic clausification is enabled. To test its effects, we enabled it in the best configuration from Figure 2, DCS−AVATAR. With this change, Zipperposition solves 1744 problems, with 36 unique solutions with respect to other configurations in Figure 2. Boolean instantiation is the only way Zipperposition 2 can solve higher-order problems requiring advanced Boolean reasoning such as `DAT056^2`.

5 Enumerating Infinitely Branching Inferences

As an optimization and to simplify the implementation, Leo-III [35] and Vampire [7] compute only a finite subset of conclusions of inferences that require enumerating a CSU. Not only is this a source of incompleteness, but choosing the cardinality of the computed subset is a difficult heuristic choice. Small sets possibly lead to missing the unifier necessary for the proof, whereas large sets make the prover spend a long time in the unification procedure, generate useless clauses, and possibly get bogged down in the wrong parts of the search space.

We propose a modification to the given clause procedure to seamlessly interleave unifier computation and proof state exploration. Given a complete unification procedure, which may yield infinite streams of unifiers, our modification fairly enumerates all conclusions of inferences relying on elements of a CSU. Under some reasonable assumptions, it behaves exactly like the standard given clause procedure on purely first-order problems. We also describe heuristics that help achieve a similar performance as when using incomplete, terminating unification procedures without sacrificing completeness.

Given that it is undecideable whether there is a next CSU element in a stream of unifiers, the request for the next conclusion might not terminate, effectively bringing the theorem prover to a halt. Our modified given clause procedure expects that unification procedure returns a lazily evaluated stream [30, Sect. 4.2], each element of which is either $\emptyset$ or a singleton set containing a unifier. To avoid getting stuck waiting for a unifier that may not exist, the unification procedure should return $\emptyset$ after it performs a number of operations without finding a unifier.

The complete unification procedure by Vukmirović et al. [46] returns such a stream. Other procedures such as that of Huet [19] and of Jensen and Pietrzykowski [20] can easily be adapted to meet this requirement. Based on the stream of unifiers interspersed with $\emptyset$, we can construct a stream of inferences similarly interspersed with $\emptyset$ of which any finite prefixes can be computed in finite time.

To support such streams in the given clause procedure, we extend it to represent the proof state not only by the active (A) and passive (P) clause sets, but also by the priority queue Q containing the inference streams. Elsewhere [3] Bentkamp et al. briefly described an older version of this extension. Here we present a newer version in more detail, including heuristics to postpone unpromising streams. The pseudocode of the modified procedure is given in Figure 3.

Initially, all input clauses are put into P, and A and Q are empty. Unlike the standard given clause procedure, we represent inference results as clause streams. Only the first element of the stream is inserted into P, and the remaining of the stream is stored in Q with some initial positive integer penalty p.

The functions Probe and ForceProbe extract some conclusions from the inference streams and store them in P. Probe heuristically chooses some clauses, whereas ForceProbe is used when P is empty to find one clause if previous probes failed to produce any clauses, as an emergency measure. By choosing clauses from several streams at the same time, we give more freedom to the prover's existing clause selection heuristics, which select a clause from P.

Probe has two modes of operation, controlled by a parameter m (by default, $m = 70$). In every mth invocation, Probe extracts conclusions from an increasing number of oldest streams. This amounts to dovetailing, which achieves fairness. In the remaining invocations, Probe selects the n highest priorty streams (by default, $n = 10$) and extracts elements from each stream—or *probes* each stream—discarding $\emptyset$ values, until a clause is found or a limit k (by default, $k = 20$) is reached on the number of discarded $\emptyset$s. Setting $k > 1$ ensures that promising streams are given a reasonable chance to produce a clause, even if they rely on a complicated unifier. Finally, each probed stream S is put back into Q with an updated priority. The new priority is based on the number of times S was chosen for probing and whether it eventually produced a clause.

The Probe and ForceProbe functions are invoked by GivenClause, which forms the body of the saturation loop. Compared with the standard given clause procedure, the differences are as follows: First, the proof state also includes Q in addition to P and A. Second, new inferences involving the given clause are put into Q instead of being performed immediately. Third, inferences in Q are periodically performed lazily to fill P.

The GivenClause function eagerly stores the first element of a new inference stream in P to imitate the standard given clause procedure. If the underlying unification procedure behaves like the standard first-order unification algorithm on the first-order fragment of higher-order logic, our given clause procedure coincides with the standard one. For example, the unification procedure by Vukmirović et al. terminates on the first-order and other fragments [28]. Moreover, for problems outside these fragments, it immediately returns $\emptyset$ to avoid computing complicated unifiers eagerly.

Evaluation and Discussion. When the unification procedure of Vukmirović et al. was implemented in Zipperposition, it was observed that Zipperposition is

```
function PROBE($Q$, $i$)
    if $i$ mod $m = 0$ then $Chosen \leftarrow$ pop $\lfloor i/m \rfloor$ oldest streams from $Q$
    else $Chosen \leftarrow$ pop $n$ highest-priority streams from $Q$
    $Collected \leftarrow \emptyset$
    for $S$ in $Chosen$ do
        $num\_probes \leftarrow 0$
        $Concl \leftarrow \emptyset$
        while $num\_probes < k$ and $Concl = \emptyset$ do
            $Concl \leftarrow$ pop and evaluate the first element of stream $S$
            $num\_probes \leftarrow num\_probes + 1$
        $Collected \leftarrow Collected \cup Concl$
        if $S$ is not empty then
            Add $S$ to $Q$ with updated priority
    return $Collected$

function FORCEPROBE($Q$)
    $Concl \leftarrow \emptyset$
    while $Q$ is not empty and $Concl = \emptyset$ do
        $streams \leftarrow$ pop all streams stored in $Q$
        while $streams$ is not empty and $Concl = \emptyset$ do
            $S \leftarrow$ pop the first element of $streams$
            $Concl \leftarrow$ pop and evaluate the first element of stream $S$
            if $S$ is not empty then
                Add $S$ to $Q$ with updated priority
        Put all remaining $S \in streams$ to $Q$
    if $Q$ and $Concl$ are empty then return (Satisfiable, $Concl$)
    else return (Unknown, $Concl$)

function GIVENCLAUSE($P$, $A$, $Q$)
    $status \leftarrow$ Unknown
    $i \leftarrow 0$
    while $status =$ Unknown do
        if $P$ is not empty then
            $C \leftarrow$ pop a chosen clause from $P$ and simplify it
            if $C$ is an empty clause then
                $status \leftarrow$ Unsatisfiable
            else
                $A \leftarrow A \cup \{C\}$
                for $(S, p)$ in streams of inferences between $C$ and $D \in A$ do
                    $Concl \leftarrow$ pop the first element of $S$
                    $P \leftarrow P \cup Concl$
                    Add $S$ to $Q$ with penalty $p$
                $i \leftarrow i + 1$
                $P \leftarrow P \cup$ PROBE$(Q, i)$
        else
            $(status, Concl) \leftarrow$ FORCEPROBE$(Q)$
            $P \leftarrow P \cup Concl$
    return $status$
```

Fig. 3: Our modified given clause procedure

the only competing higher-order prover that solves all Church numeral problems from the TPTP, never spending more than 5 seconds on the problem [46].

Consider the TPTP problem `NUM800^1`, which claims that there exists a function F such that $F\,\mathsf{c}_1\,\mathsf{c}_2 \approx \mathsf{c}_2 \wedge F\,\mathsf{c}_2\,\mathsf{c}_3 \approx \mathsf{c}_6$, where c_n abbreviates the nth Church numeral, $\lambda s\, z.\, s^n(z)$, in which the exponent denotes iterated application. To solve the problem, it suffices to take F to be the multiplication operator $\lambda x\, y\, s\, z.\, x\,(y\, s)\, z$. However, this unifier is only one out of many available for each occurrence of F.

In an independent evaluation setup on the same set of 2606 problems used in this paper, Vukmirović et al. compared a complete and a terminating, pragmatic variant of the unification procedure [46, Sect. 7]. The pragmatic variant was used directly—all the inference conclusions were put immediately in P, bypassing Q. The complete variant, which relies on possibly infinite streams and is considerably more prolific, solved only 15 problems less than the most competitive pragmatic variant. Furthermore, it solved 19 problems not solved by the pragmatic variant. This shows that our given clause procedure, with its heuristics, allows the prover to defer exploring less promising branches of the unification and uses the full potential of complete higher-order unifier search to solve unification problems that cannot be solved by a crippled unification procedure.

Among the competing higher-order theorem provers, only Satallax uses infinitely branching calculus rules. It maintains a queue of *commands* that contain instructions on how to create a successor state in the tableaux. One of the commands describes infinite enumeration of all closed terms of a given function type. Each execution of this command makes progress in the enumeration. Unlike evaluation of streams representing elements of CSU, each execution of a command is guaranteed to make progress in enumerating the next closed functional term, so there is no need to ever return $\emptyset$.

6 Controlling Prolific Rules

To support higher-order features such as function extensionality and quantification over functions, many refutationally complete calculi employ highly prolific rules. For example, λ-superposition extends first-order superposition with a rule FluidSup [3] that almost always applies to two clauses if one of them contains a term of the form $F\,\overline{s}_n$, where $n > 0$. We describe three mechanisms to keep rules like these under control.

First, *we limit applicability of the prolific rules.* In practice, it often suffices to apply prolific higher-order rules only to initial or shallow clauses—clauses with a shallow derivation depth. Thus, we added an option to forbid the application of a rule if the derivation depth of any premise exceeds a limit.

Second, *we penalize the streams of expensive inferences.* The penalty of each stream is given an initial value based on characteristics of the inference premises such as their derivation depth. For prolific rules such as FluidSup, we increment this value by a parameter p. Penalties for less prolific variants of this rule, such as DupSup [3], are increased by fraction of p (e.g., $\lfloor p/3 \rfloor$).

base	`BAVN`	`PL`	`PSAV`	`PHOS`	`BNF`	`PDAV`
1638	**1640**	1637	1637	1632	1594	1520

Fig. 4: Effect of the priority function on the performance

Third, *we defer the selection of prolific clauses.* To select the given clause, most saturating provers evaluate clauses according to some criteria and select the clause with the lowest evaluation. For this choice to be efficient, passive clauses are organized into a priority queue ordered by their evaluations. Following E [33], Zipperposition maintains multiple queues, ordered by different evaluations, that are visited in a round-robin fashion. It also uses E's two-layer evaluation functions, a variant of which has recently been implemented in Vampire [16]. The two layers are *clause priority* and *clause weight*. Clauses with higher priority are preferred, and the weight is used for tie-breaking. Intuitively, the first layer crudely separates clauses into priority classes, whereas the second one uses heuristic weights to prefer clauses within a class. To control the selection of prolific clauses, we introduce new clause priority functions that take into account features specific to higher-order clauses.

The first new priority function `PreferHOSteps` (`PHOS`) assigns a higher priority if rules specific to λ- or combinatory superposition were used in the clause derivation. Since most of the other clause priority functions tend to defer higher-order clauses, having a clause queue that prefers results of higher-order inferences might be necessary to find a proof more efficiently. A simpler function that tries to discover clauses with higher-order properties is `PreferLambda` (`PL`), which prefers clauses that contain λ-abstractions.

We also introduce the priority function `ByNormalizationFactor` (`BNF`), based on an observation that a higher-order inference that applies a complicated substitution to a clause is usually followed by a $\beta\eta$-normalization step. If $\beta\eta$-normalization greatly reduces the size of a clause, it is likely that this substitution simplifies the clause (e.g., by removing arguments of a variable). Thus, this function prefers clauses that were produced by $\beta\eta$-normalization, and among those it prefers the ones with larger size reductions.

Another new priority function is `PreferShallowAppVars` (`PSAV`). It prefers clauses with lower depths of the deepest occurrence of an applied variable, i.e., $C[X\,\mathsf{a}]$ is preferred over $C[\mathsf{f}\,(X\,\mathsf{a})]$. This function tries to curb the explosion of both λ- and combinatory superposition: Applying a substitution to a top-level applied variable often reduces this applied variable to a term with a constant head, which likely results in a less explosive clause. Among the functions that rely on properties of applied variables we implemented `PreferDeepAppVars` (`PDAV`), which returns the priority opposite of `PSAV`, and `ByAppVarNum` (`BAVN`), which prefers clauses with fewer occurrences of applied variables.

Evaluation and Discussion. In the base configuration, Zipperposition visits several clause queues, one of which uses the constant priority function. To eva-

base	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$
1638	1610	1612	1618	1621	1619

Fig. 5: Effect of the FluidSup penalty on the performance

luate the new priority functions, we replaced the queue ordered by the constant priority with the queue ordered by one of the new functions, leaving the clause weight intact. The results are shown in Figure 4. It shows that the expensive priority functions `PHOS` and `BNF`, which require inspecting the proof of clauses, hardly help. Simple functions such as `PL` are more effective: Compared with *base*, `PL` loses one solution overall but solves 22 new problems.

FluidSup is disabled in *base* because it is so explosive. To test if different inference stream queue priorities make a difference on the success rate, we enabled FluidSup and used different priority increments p for FluidSup inference queues. The results are shown in Figure 5. As expected, giving a low penalty to FluidSup is detrimental to performance. However, as the column for $p = 16$ shows, we should not give too high a penalty increase either, since that delays useful FluidSup inferences. Interestingly, even though configuration with $p = 1$ solves the least problems overall, it solves 7 problems not solved by *base*, which is more than any other configuration we tried.

7 Controlling the Use of Backends

Cooperation with efficient off-the-shelf first-order theorem provers is an essential feature of higher-order theorem provers such as Leo-III [35, Sect. 4.4], Satallax [10], and Sledgehammer [9]. Those provers invoke first-order backends repeatedly during a proof attempt and spend a substantial amount of time in backend collaboration. Since λ-superposition generalizes a highly efficient first-order calculus, we expect that future efficient λ-superposition implementations will not benefit much from backends. Experimental provers such as Zipperposition, on the other hand, can gain a lot. We present some techniques for controlling the use of backends.

In his thesis [35, Sect. 6.1], Steen extensively evaluates the effects of using different first-order backends on the performance of Leo-III. His results suggest that adding only one backend already substantially improves the performance. To reduce integration effort related to using multiple backends, we chose Ehoh [48] as our single backend. Ehoh is an extension of the highly optimized superposition prover E [33] with support for higher-order features such as partial application, applied variables, and Booleans. Ehoh provides the efficiency of E while easing the translation from full higher-order logic: The only missing syntactic feature is λ-abstraction. On the other hand, Ehoh's higher-order reasoning capabilities are limited. Its unification algorithm is essentially first-order and it cannot synthesize λ-abstractions or instantiate predicate variables.

In a departure from Leo-III and other cooperative provers, instead of regularly invoking the backend, we invoke it at most once during a run of the prover. This makes sense because most competitive higher-order provers use a portfolio mode in which many configurations are run for a short time, and we want to leave enough time for native higher-order reasoning. Moreover, multiple backends invocations tend to be wasteful, because each invocation starts with no knowledge of the previous ones. Maybe this could be addressed in the future.

Only a subset of the available clauses are translated and sent to Ehoh. The choice of clauses is crucial. Let I be the set of clauses representing the input problem. Let M denote the union of the current active and passive sets. Let M_{ho} denote the subset of M that contains only clauses that were derived using at least one λ-superposition-specific inference rule. We order the clauses in M_{ho} by increasing derivation depth, using syntactic weight to break ties. Then, we choose all clauses in I and the first n clauses from M_{ho} for use with the backend reasoner. From all derived clauses in M, we include only the clauses from M_{ho} because clauses derived using first-order rules can also be derived by Ehoh if necessary. We also expect large clauses with deep derivations to be less useful.

The remaining step is the translation of λ-abstractions. We support two translation methods: λ-lifting [21] and Curry's SKBCI combinator translation [44]. For SKBCI, we exclude the combinator definition axioms, because they are very explosive [8]. A third mode simply omits clauses containing λ-abstractions.

Evaluation and Discussion. Several parameters control Zipperposition's use of the Ehoh backend. These include the CPU time allotted to Ehoh, Ehoh's own parameters, the point when Ehoh is invoked, the size of M_{ho}, and the λ translation method. We fix the time limit to 5 s and use Ehoh in *auto* mode, and focus on the last three parameters. In *base*, collaboration with Ehoh is disabled.

Ehoh is invoked after $p \cdot t$ CPU seconds, where $0 \leq p < 1$ and t is the total CPU time allotted to Zipperposition. Figure 6 shows the effect of varying p when the size of M_{ho} is fixed to 32 and λ-lifting is used. The evaluation confirms that using a highly optimized backend such as Ehoh greatly improves the performance of a less optimized prover such as Zipperposition. The figure indicates that it is preferable to invoke the backend early. We have indeed observed that if the backend is invoked late, small clauses with deep derivation tend to be present by then. These clauses might have been used to delete important shallow clauses already. But due to their derivation depth, they will not be translated. In such situations, it is better to invoke the backend before the important clauses are deleted.

Figure 7 quantifies the effects of the λ-abstraction translation methods. We fixed $p = 0.25$ and $|M_{\text{ho}}| = 32$. The clear winner is λ-lifting. Remarkably, omitting clauses with λ-abstractions performs comparably to SKBCI combinators.

Figure 8 reveals how the size of M_{ho} affects performance, with $p = 0.25$ and λ-lifting. We find that including a small number of higher-order clauses with the lowest weight performs better than including a large number of such clauses.

base	$p = 0.1$	$p = 0.25$	$p = 0.5$	$p = 0.75$
1638	**1936**	1935	1934	1923

Fig. 6: Effect of the backend invocation point

base	lifting	SKBCI	omitted
1638	**1935**	1867	1855

Fig. 7: Effect of λ-abstraction translation method

base	$m = 16$	$m = 32$	$m = 64$	$m = 128$	$m = 256$	$m = 512$
1638	1936	1935	**1939**	1928	1925	1912

Fig. 8: Effect of the size m of M_{ho}

8 Comparison with Other Provers

Raw evaluation data of the previous experiments shows that different choices of parameters lead to noticeably different sets of solved problems. In an attempt to use Zipperposition 2 to its full potential, we have created a portfolio mode that runs up to 50 configurations in parallel during the allotted time. To provide some context, we compare Zipperposition 2 with the latest versions of all higher-order provers that competed at CASC-J10: CVC4 1.8 [2], Leo-III 1.5 [37], Satallax 3.5 [10], and Vampire 4.5 [25]. Note that Vampire's higher-order schedule is optimized for running on a single core.

We use the same set of 2606 monomorphic higher-order TPTP 7.2.0 problems as elsewhere in this paper, but we try to replicate the CASC setup more faithfully. CASC-J10 was run on 8-core CPUs with a 120 s wall-clock limit and a 960 s CPU limit. As we run our experiments on 4-core CPUs, we set the wall-clock limit to 240 s and keep the same CPU limit. Leo-III, Satallax, and Zipperposition are cooperative provers. We also run them in uncooperative mode, without their backends, to measure their intrinsic strength. Figure 9 summarizes the results.

Among the cooperative provers, Zipperposition is the one that depends the least on its backend, and its uncooperative mode is only one problem behind Satallax *with a backend*. This confirms our hypothesis that λ-superposition is a suitable basis for automatic higher-order reasoning. The increase in performance due to the addition of an efficient backend suggests that the implementation of this calculus in a modern first-order superposition prover such as E or Vampire would achieve markedly better results. Moreover, we believe that there are still techniques inspired by tableaux, SAT solving, and SMT solving that could be adapted and integrated in saturation provers.

9 Discussion and Conclusion

Back in 1994, Kohlhase [24, Sect. 1.3] was optimistic about the future of higher-order automated reasoning:

	uncoop	coop
CVC4	1810	–
Leo-III	1641	2108
Satallax	2089	2224
Vampire	2096	–
Zipperposition	2223	**2307**

Fig. 9: Comparison with other competing higher-order theorem provers

> The author believes that the obstacles to proof search intrinsic to higher-order logic may well be compensated by the greater expressive power of higher-order logic and by the existence of shorter proofs. Thus higher-order automated theorem proving will be practically as feasible as first-order theorem proving is now as soon as the technological backlog is made up.

For higher-order superposition, the backlog consisted of designing calculus extensions, heuristics, and algorithms that mitigate its weaknesses. In this paper, we presented such enhancements, justified their design, and evaluated them. We explained how each weak point in the higher-order proving pipeline could be improved, from preprocessing to reasoning with formulas, to delaying unpromising or explosive inferences, to invoking a backend. Our evaluation indicates that higher-order superposition is now the state of the art in higher-order reasoning.

Higher-order extensions of first-order superposition have been described from a theoretical perspective by Bentkamp et al. [3, 5] and Bhayat and Reger [7, 8]. They introduced proof calculi, proved them refutationally complete, and discussed some optional rules, but they hardly dicussed the practical aspects of higher-order superposition. In contrast, there is a vast literature on practical aspects of first-order reasoning using superposition and related calculi. The literature evaluates various procedures and techniques [18, 32], literal and term order selection functions [17], and clause evaluation functions [16, 34], among others. Our work joints the select club of papers that focus on practical aspects of higher-order reasoning [6, 13, 36, 49].

As a next step, we plan to implement the described techniques in Ehoh [48], the λ-free higher-order extension of E. We expect the resulting prover to be substantially more efficient than Zipperposition. Moreover, we want to investigate the proofs found by provers such as CVC4 and Satallax but missed by Zipperposition. Finding the reason behind why Zipperposition fails to solve a problem will likely result in useful new techniques.

Acknowledgment. We are grateful to the maintainers of StarExec for letting us use their service. Ahmed Bhayat and Giles Reger guided us through details of Vampire 4.5. Ahmed Bhayat, Michael Färber, and Mathias Fleury suggested textual or typesetting improvements. We thank them all. Vukmirović, Bentkamp, and Blanchette's research has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Blanchette and Nummelin's research has received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

References

1. Backes, J., Brown, C.E.: Analytic tableaux for higher-order logic with choice. J. Autom. Reason. **47**(4), 451–479 (2011)
2. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV. LNCS, vol. 6806, pp. 171–177. Springer (2011)
3. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P., Waldmann, U.: Superposition with lambdas. J. Autom. Reason. To appear
4. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P.: Superposition for full higher-order logic (2021), submitted to CADE-28
5. Bentkamp, A., Blanchette, J.C., Cruanes, S., Waldmann, U.: Superposition for lambda-free higher-order logic. In: IJCAR. LNCS, vol. 10900, pp. 28–46. Springer (2018)
6. Benzmüller, C., Sorge, V., Jamnik, M., Kerber, M.: Can a higher-order and a first-order theorem prover cooperate? In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS, vol. 3452, pp. 415–431. Springer (2004)
7. Bhayat, A., Reger, G.: Restricted combinatory unification. In: CADE. LNCS, vol. 11716, pp. 74–93. Springer (2019)
8. Bhayat, A., Reger, G.: A combinator-based superposition calculus for higher-order logic. In: IJCAR (1). LNCS, vol. 12166, pp. 278–296. Springer (2020)
9. Böhme, S., Nipkow, T.: Sledgehammer: Judgement Day. In: IJCAR 2010. LNCS, vol. 6173, pp. 107–121. Springer (2010)
10. Brown, C.E.: Reducing higher-order theorem proving to a sequence of SAT problems. J. Autom. Reason. **51**(1), 57–77 (2013)
11. Cruanes, S.: Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond. Ph.D. thesis, École polytechnique (2015)
12. Czajka, L., Kaliszyk, C.: Hammer for coq: Automation for dependent type theory. J. Autom. Reason. **61**(1-4), 423–453 (2018)
13. Färber, M., Brown, C.E.: Internal guidance for Satallax. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS, vol. 9706, pp. 349–361. Springer (2016)
14. Filliâtre, J., Paskevich, A.: Why3 - where programs meet provers. In: ESOP. LNCS, vol. 7792, pp. 125–128. Springer (2013)
15. Ganzinger, H., Stuber, J.: Superposition with equivalence reasoning and delayed clause normal form transformation. In: CADE. LNCS, vol. 2741, pp. 335–349. Springer (2003)
16. Gleiss, B., Suda, M.: Layered clause selection for theory reasoning - (short paper). In: IJCAR (1). Lecture Notes in Computer Science, vol. 12166, pp. 402–409. Springer (2020)
17. Hoder, K., Reger, G., Suda, M., Voronkov, A.: Selecting the selection. In: IJCAR. Lecture Notes in Computer Science, vol. 9706, pp. 313–329. Springer (2016)

18. Hoder, K., Voronkov, A.: Comparing unification algorithms in first-order theorem proving. In: KI. LNCS, vol. 5803, pp. 435–443. Springer (2009)
19. Huet, G.P.: A unification algorithm for typed lambda-calculus. Theor. Comput. Sci. **1**(1), 27–57 (1975)
20. Jensen, D.C., Pietrzykowski, T.: Mechanizing *omega*-order type theory through unification. Theor. Comput. Sci. **3**(2), 123–171 (1976)
21. Johnsson, T.: Lambda lifting: Treansforming programs to recursive equations. In: FPCA. LNCS, vol. 201, pp. 190–203. Springer (1985)
22. Kaliszyk, C., Urban, J.: Hol(y)hammer: Online ATP service for HOL light. Math. Comput. Sci. **9**(1), 5–22 (2015)
23. Knuth, D.E., Bendix, P.B.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263 – 297. Pergamon (1970)
24. Kohlhase, M.: A mechanization of sorted higher-order logic based on the resolution principle. Ph.D. thesis, Saarland University, Saarbrücken, Germany (1994)
25. Kovács, L., Voronkov, A.: First-order theorem proving and vampire. In: CAV. LNCS, vol. 8044, pp. 1–35. Springer (2013)
26. McCune, W., Wos, L.: Otter—the CADE-13 competition incarnations. J. Autom. Reason. **18**(2), 211–220 (1997)
27. Nipkow, T.: Functional unification of higher-order patterns. In: Best, E. (ed.) LICS '93. pp. 64–74. IEEE Computer Society (1993)
28. Nipkow, T.: Functional unification of higher-order patterns. In: LICS. pp. 64–74. IEEE Computer Society (1993)
29. Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning (in 2 volumes), pp. 335–367. Elsevier and MIT Press (2001)
30. Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
31. Paulson, L.C., Blanchette, J.C.: Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In: IWIL@LPAR. EPiC Series in Computing, vol. 2, pp. 1–11. EasyChair (2010)
32. Reger, G., Suda, M., Voronkov, A.: Playing with AVATAR. In: CADE. LNCS, vol. 9195, pp. 399–415. Springer (2015)
33. Schulz, S., Cruanes, S., Vukmirović, P.: Faster, higher, stronger: E 2.3. In: CADE. LNCS, vol. 11716, pp. 495–507. Springer (2019)
34. Schulz, S., Möhrmann, M.: Performance of clause selection heuristics for saturation-based theorem proving. In: IJCAR. LNCS, vol. 9706, pp. 330–345. Springer (2016)
35. Steen, A.: Extensional paramodulation for higher-order logic and its effective implementation Leo-III. Ph.D. thesis, Free University of Berlin, Dahlem, Germany (2018)
36. Steen, A., Benzmüller, C.: There is no best β-normalization strategy for higher-order reasoners. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR-20. LNCS, vol. 9450, pp. 329–339. Springer (2015)
37. Steen, A., Benzmüller, C.: The higher-order prover leo-iii. In: IJCAR. LNCS, vol. 10900, pp. 108–116. Springer (2018)
38. Stump, A., Sutcliffe, G., Tinelli, C.: Starexec: A cross-community infrastructure for logic solving. In: IJCAR. LNCS, vol. 8562, pp. 367–373. Springer (2014)
39. Sultana, N., Blanchette, J.C., Paulson, L.C.: LEO-II and satallax on the sledgehammer test bench. J. Appl. Log. **11**(1), 91–102 (2013)
40. Sutcliffe, G.: The CADE ATP System Competition - CASC. AI Magazine **37**(2), 99–101 (2016)

41. Sutcliffe, G.: The TPTP problem library and associated infrastructure - from CNF to th0, TPTP v6.4.0. J. Autom. Reason. **59**(4), 483–502 (2017)
42. Sutcliffe, G.: The CADE-27 automated theorem proving system competition—CASC-27. AI Commun. **32**(5-6), 373–389 (2019)
43. Sutcliffe, G.: The 10th IJCAR automated theorem proving system competition—CASC-J10. AI Communications (2021), to appear
44. Turner, D.A.: Another algorithm for bracket abstraction. J. Symb. Log. **44**(2), 267–270 (1979)
45. Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: CAV. LNCS, vol. 8559, pp. 696–710. Springer (2014)
46. Vukmirović, P., Bentkamp, A., Nummelin, V.: Efficient full higher-order unification. In: FSCD. LIPIcs, vol. 167, pp. 5:1–5:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
47. Vukmirović, P., Nummelin, V.: Boolean reasoning in a higher-order superposition prover. In: PAAR (2020)
48. Vukmirović, P., Blanchette, J.C., Cruanes, S., Schulz, S.: Extending a brainiac prover to lambda-free higher-order logic. In: TACAS (1). LNCS, vol. 11427, pp. 192–210. Springer (2019)
49. Wisniewski, M., Steen, A., Kern, K., Benzmüller, C.: Effective normalization techniques for HOL. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS, vol. 9706, pp. 362–370. Springer (2016)